# REST

*Angelo Di Iorio*

*Università di Bologna*

*(dal materiale del Prof. Fabio Vitali)*

# REST

- REST è l'acronimo di ***REpresentional State Transfer***, ed è il **modello architetturale** che sta dietro al World Wide Web e in generale dietro alle applicazioni web "ben fatte" secondo i progettisti di HTTP.
- Applicazioni non REST si basano sulla generazione di un API che specifica le funzioni messe a disposizione dell'applicazione, e alla creazione di un'interfaccia ***indipendente*** dal protocollo di trasporto e ad essa completamente ***nascosta***.
- Viceversa, un'applicazione REST si basa fondamentalmente sull'uso dei protocolli di trasporto (HTTP) e di naming (URI) per generare interfacce ***generiche*** di interazione con l'applicazione, e **fortemente connesse** con l'ambiente d'uso.

# API Web

- Un'API Web descrive un interfaccia HTTP che permette ad applicazioni remote di utilizzare i servizi di dell'applicazione
- Queste possono essere:
  - Applicazioni automatiche che utilizzano i dati della mia applicazione
  - Applicazioni Web che mostrano all'utente un menù di opzioni, magari anche un form, e gli permettono di eseguire un'azione sui dati della mia applicazione.
- Idealmente, il programmatore dell'applicazione server-side non dovrebbe neanche sapere se la richiesta gli arriva da un modulo interno dell'applicazione o da una richiesta esterna via HTTP

# Il modello CRUD

- Un pattern tipico delle applicazioni di trattamento dei dati
- Ipotizza che tutte le operazioni sui dati siano una di:
  - **Create** (inserimento nel database di un nuovo record)
    - Crea un cliente il cui nome è "Rossi SpA", il telefono "051 654321", la città "Bologna" e restituisce il codice identificatore che è 4123.
  - **Read** (accesso in lettura al database)
    - ***individuale***: dammi la scheda del cliente con id=4123,
    - ***contenitore***: dammi la lista dei clienti la cui proprietà *città* è uguale al valore "Bologna"
  - **Update**
    - Cambia il numero di telefono del cliente il cui id=4123 in "051 123456"
  - **Delete**
    - Rimuovi dal database il cliente con id=4123

# REpresentational State Transfer

L'architettura REST si basa su quattro punti :

1. Definire **risorsa** ogni concetto rilevante dell'applicazione Web
2. Associargli un **URI** come l'**identificatore** e selettore primario
3. Usare i verbi HTTTP per esprimere ogni **operazione** dell'applicazione secondo il modello CRUD:
   - creazione di un nuovo oggetto (metodo PUT)
   - visualizzazione dello stato della risorsa (metodo GET)
   - cambio di stato della risorsa (metodo POST)
   - cancellazione di una risorsa (metodo DELETE)
4. Esprimere in maniera parametrica ogni **rappresentazione dello stato interno della risorsa**, personalizzabile dal richiedente attraverso un **Content Type** preciso

# Approccio Web Service tradizionale

- Nelle applicazioni non REST, progettate secondo il modello tradizionale SOAP, esiste un intermediario di messaggi che:
  - raccoglie una richiesta via HTTP
  - ne interpreta i parametri
  - chiama la funzione interna corretta con i parametri corretti
  - riceve il valore di ritorno
  - lo ritrasmette al cliente originale.
- La richiesta è indirizzata a questo intermediario e contiene nel body tutte le informazioni necessarie per soddisfarla
- Il protocollo HTTP è usato solo per trasferire le informazioni ma in modo trasparente

# Esempio: getClient(clientId)

```
POST dispatch HTTP/1.1

Host: http://www.sito.com:80
Content-Type: text/xml; charset=utf-8
Content-length: 474

<?xml version="1.0"?>

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/
soap/envelope/" xmlns:xsi="http://www.w3.org/1999/
XMLSchema-instance">
  <SOAP:Body>
    <m:getClient xmlns:m="http
      <clientId xsi:type="xsd:token
    </m:getClient>
  </SOAP:Body>
</SOAP:Envelope>
```

Il metodo è sempre POST

L' URI è sempre lo stesso

Il body SOAP contiene un elemento che si chiama come la funzione e ha una lista esplicita di parametri

# La risposta: getClient(clientId)

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 499
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2001 05:05:04 GMT

<?xml version="1.0"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
   <S:Body>
      <m:getClientResponse xmlns:m="http://www.myApp.org/">
         <Result xsi:type="xsd:string">South Dakota</Result>
         </m:getClientResponse>
      </S:Body>
   </S:Envelope>
```

La risposta è sempre uguale

# REST

- L'architettura REST si basa sui tre presupposti del Web, sui quali il sistema è stato progettato sin dall'inizio:
  - Che venga definita come *risorsa* ogni entità, ed associato ad essa un URI come identificatore (*il **nome***)
  - Che ogni interazione con un'applicazione Web sia esprimibile con un metodo HTTP (*il **verbo***)
  - Che ciò che viene scambiato tra gli attori dell'interazione sia una rappresentazione di uno stato della risorsa specificato attraverso un Content-Type (*il **formato***).
- REST infatti non è un nuovo protocollo, ma un modo di strutturare le applicazioni per sfruttare pienamente le caratteristiche di HTTP

# Esempio REST: crea cliente

Il metodo specifica l'operazione eseguita

```
PUT clients/1234 HTTP/1.1
Host: http://www.sito.com:80
Content-Type: text/xml; charset=utf-8
Content-length: 474

<client xmlns:m="http://www.myAp
   <nome>Rossi S.p.A.</nome>
   <tel>051 654321</t
   <citta>Bologna</citta>
</client>
```

L' URI dell'oggetto coinvolto

Il body contiene una rappresentazione (in questo caso XML) dell'oggetto da creare

# Esempio REST: aggiorna cliente

Il metodo PUT è usato sia per creare che per sostituire una risorsa

```
PUT clients/1234 HTTP/1.1
Host: http://www.sito.com:80
Content-Type:application/json; charset=utf-8
Content-length: 474

{
    "nome": "Rossi S.p.A.",
    "tel" : "051 654321",
    "citta" : "Bologna"
}
```

Il body contiene una rappresentazione JSON dell'oggetto da sovrascrivere

# Individui e collezioni

- REST identifica due concetti fondamentali: **individui** e **collezioni**
  - un cliente vs. l'insieme di tutti i clienti
  - un esame vs. l'insieme di tutti gli esami superati
  - …
- Fornisce URI ad entrambi
- Ogni operazione avviene su uno e uno solo di questi concetti.
- Su entrambi si possono eseguire operazioni CRUD. A seconda della combinazione di verbi e risorse otteniamo l'insieme delle operazioni possibili.
- Ciò che passa come corpo di una richiesta e/o risposta NON E' la risorsa, ma una *rappresentazione* della risorsa, di cui gli attori si servono per portare a termine l'operazione.

# Gerarchie

- Le collezioni possono "contenere" individui o altre collezioni
- E' consigliabile strutturare gli URI in modo gerarchico, per esplicitare queste relazioni
- API più leggibile e *routing* semplificato in molti framework di sviluppo
- Ad esempio:
  - Tutti i clienti

    **`/clients/`**
  - Il cliente 1234

    **`/clients/1234/`**
  - Tutti gli ordini del cliente 1234

    **`/clients/1234/orders/`**

# Linee guida degli URI in REST

Le **collezioni** sono intrinsecamente **plurali**. Gli **individui** sono intrinsecamente **singolari**.

Le collezioni debbono essere visivamente distinguibili dagli individui. Per questo usiamo un termine plurale ***e*** uno slash in fondo all'URI

| URI | Rappresentazione |
|---|---|
| `/customers/` | collezione dei clienti |
| `/customers/abc123` | cliente con id=abc123 |
| `/customers/abc123/` | collezione delle sotto-risorse del cliente con id=abc123 (es. indirizzi, telefoni,ecc.) |
| `/customers/abc123/addresses/1` | primo indirizzo del cliente con id=abc123 |

# Filtri e search negli URI REST

Un filtro genera un **sottoinsieme** specificato attraverso una regola di qualche tipo. La gerarchia permette di specificare i tipi più frequenti e rilevanti di filtro.

Altrimenti si usa la parte query dell'URI di una collezione:

| URI | Rappresentazione |
|---|---|
| `/regions/ER/customers/` | collezione dei clienti dell'Emilia Romagna |
| `/status/active/customers/` | collezione dei clienti attivi |
| `/customers/?tel=0511234567` oppure `/customers/?search=tel&value=0511234567` | collezione dei clienti che hanno telefono = 051 1234567 |
| `/customers/?search=sales&value=100000&op=gt` | collezione dei clienti che hanno comprato più di 100.000€ |

# Uso dei verbi HTTP in REST

Attenzione a
questa differenza!
Operazioni su collezione
O individuo

- Elencare tutti i clienti
  `GET /customers/`
- Accedere ai dati del cliente id=abc123
  `GET /customers/abc123`
- Creare un nuovo cliente (il client **non** decide l'identificatore)
  `POST /customers/`
- Creare un nuovo cliente (il client **decide** l'identificatore)
  `PUT /customers/abc123`
- Modificare (tutti) i dati del cliente id=abc123
  `PUT /customers/abc123`
- Modificare alcuni dati del cliente id=abc123
  `POST /customers/abc123/telephones/`
- Cancellare il cliente id=abc123
  `DELETE /customers/abc123`

# Semantica del POST

Nelle vecchie versioni di HTTP (ad es. RFC2616), si diceva:

- *"The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It essentially means that POSTrequest-URI should be of a collection URI."*

Nel 2014 è stata approvata una modifica e chiarificazione ad alcuni testi del documento di HTTP (RFC7231), e in particolare:

- *"The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics."*

In pratica, il POST può essere usato in una moltitudine di situazione secondo una semantica decisa localmente, purché non sovrapposta a quella degli altri verbi

# Altri consigli e linee guida

- Adottare una convenzione di denominazione coerente e chiara negli URI
- Usare gerarchie ma valutare i livelli necessari (chiarezza vs. carico sul server)
- Evitare di creare API che rispecchiano semplicemente la struttura interna di un database
- Fornire meccanismi − parametri nelle query − per filtrare e paginare le risposte
- Supportare richieste asincrone e in questo caso restituire codice HTTP 202 (Accettato ma non completato) e informazioni (URL) per accedere allo stato della risorsa

# Controllo delle versioni di un API

- E' naturale che le API siano modificate durante il loro ciclo di vita. E' fondamentale quindi **versionare** la propria **API**
- REST non fornisce linee guida stringenti ma esistono due approcci principali
- Il più usato consiste nel memorizzare il numero di versione dell'API all'inizio degli URI; paradossalmente vìola l'idea di identificare solo una risorsa nell'URI

```
http://api.miosito.com/v1/clients/
http://api.miosito.com/v1/clients/1234
```

- Alternativamente si possono usare header HTTP `Accept` e i meccanismi di *content negotiation* per specificare la versione dell'API supportata.

```
Accept: application/vnd.example.v1+json
Accept: application/vnd.example+json;version=1.0
```

- Gli URI sono più puliti ma la complessità sul client aumenta

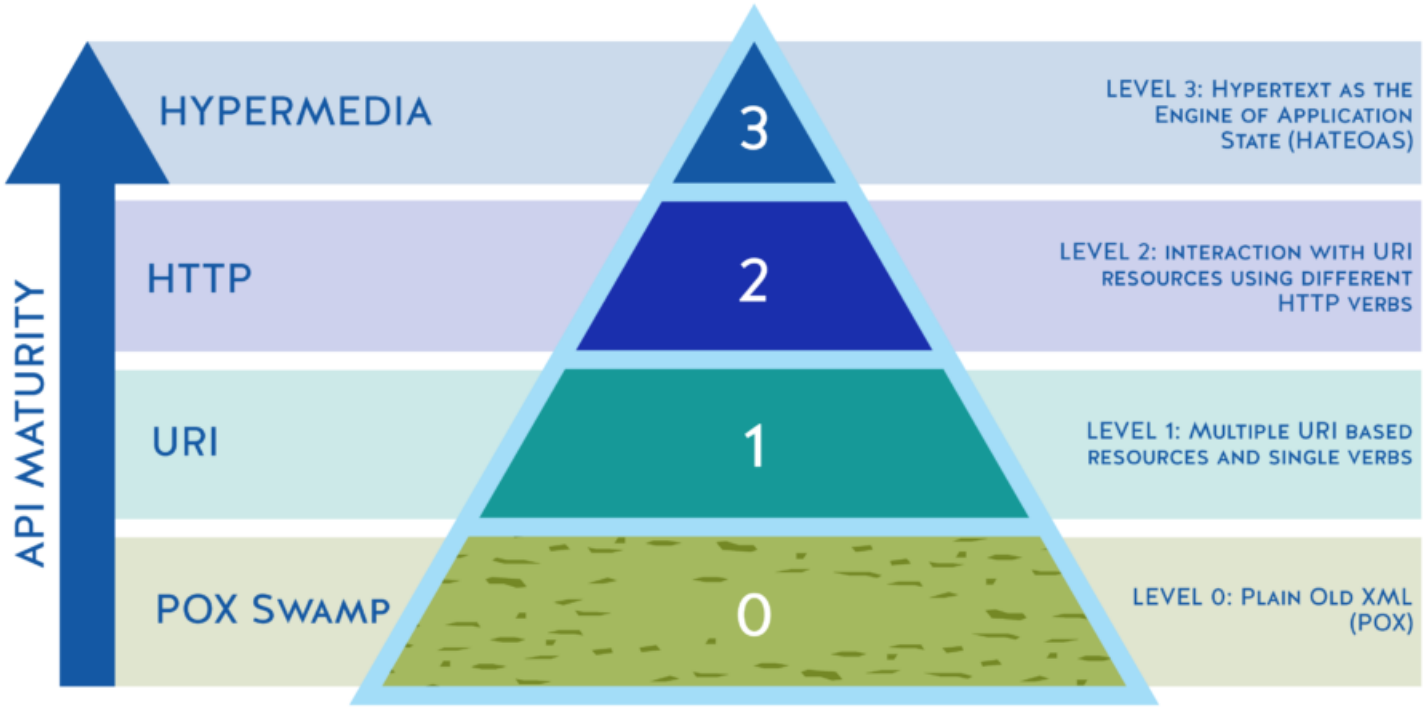
# Modelli di maturità di API

- E' molto comune trovare API che dichiarano di essere RESTful ma non rispettano completamente le linee guida del paradigma
- Diversi studiosi hanno proposto modelli per verificare "*quanto un'API è davvero REST*" e misurarne il livello di maturità

- Tra i più rilevanti il modello di Richardson, presentato per la prima volta alla conferenza QCon (Enterprise Software Development Conference) nel 2008
  - https://www.crummy.com/writing/speaking/2008-QCon/act3.html
- Un post di Martin Flower che descrive il modello di Richardson
  - https://martinfowler.com/articles/richardsonMaturityModel.html

# Modello di maturità di Richardson

- Leonard Richardson individua 4 livelli:



# Riferimenti


- T. Berners-Lee, R. Fielding, H. Frystyk, Hypertext Transfer Protocol -- HTTP/1.0, RFC 1945, May 1996
- D. Kristol, L. Montulli, HTTP State Management Mechanism, RFC 2965, October 2000
- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616, June 1999